Unified Oblivious-RAM: Improving Recursive ORAM with Locality and Pseudorandomness

Ling Ren†, Christopher Fletcher†, Xiangyao Yu†, Albert Kwon†, Marten van Dijk‡, Srinivas Devadas†

† Massachusetts Institute of Technology

‡ University of Connecticut

ABSTRACT

Oblivious RAM (ORAM) is a cryptographic primitive that hides memory access patterns to untrusted storage. ORAM may be used in secure processors for encrypted computation and/or software protection. While recursive Path ORAM is currently the most practical ORAM for secure processors, it still incurs large performance and energy overhead and is the performance bottleneck of recently proposed secure processors.

In this paper, we propose two optimizations to recursive Path ORAM. First, we identify a type of program locality in its operations to improve performance. Second, we use pseudorandom function to compress the position map. But applying these two techniques in recursive Path ORAM breaks ORAM security. To securely take advantage of the two ideas, we propose unified ORAM. Unified ORAM improves performance both asymptotically and empirically. Empirically, our experiments show that unified ORAM reduces data movement from ORAM by half and improves benchmark performance by 61% as compared to recursive Path ORAM.

1. INTRODUCTION

Oblivious RAM (ORAM) is a cryptographic primitive that conceals memory access patterns, i.e., the addresses of memory accesses. ORAM hides access patterns by continuously reshuffling the memory and translating the address of each memory access to a set of randomized memory locations. Provably, these randomized memory locations are guaranteed to be independent of the actual addresses.

ORAM was first proposed by Goldreich and Ostrovsky [8] for software protection. Their goal was to make a processor's interaction with memory *oblivious*, meaning that the interaction does not depend on the software running in the processor. Another important application of ORAM in secure processors is encrypted computation [5]. In this scenario, the user wants to do computation on an untrusted server and does not want any adversary (including the untrusted server) to learn his/her data. One solution is to have tamper-resistant secure processors on the server side. The user encrypts and sends his/her data to the secure processor, inside which the data is decrypted and computed upon. After the computation, the secure processor encrypts the results and sends them back to the user. ORAM also finds applications in areas outside processors, e.g., secure remote storage [37, 36] and proof of retrievability [2]. This paper focuses on secure processors.

While it is assumed that adversaries cannot look *inside* a tamper-resistant secure processor, we still have to protect the interaction between the processor and the external main memory. In both software protection and encrypted computation, encrypting the memory is not enough; access patterns also leak information. Zhuang et al. [45] showed that memory access patterns can leak the program's control flow graph, and therefore leak sensitive data through data-dependent conditional branches. However, most existing secure processors, including Intel's TPM+TXT [14, 42, 1], XOM [19, 20, 21] and Aegis [40, 41] have not taken measures to protect memory access patterns.

ORAM completely hides access patterns, but has long been assumed to be too expensive for processors. Only recently, ORAM was embraced by secure processors like Ascend [5, 44] and Phantom [25]. Both Ascend and Phantom use Path ORAM [38] because of its efficiency and simplicity. One problem of Path ORAM when used in processors is that it needs to store a large data structure called *Position Map* (PosMap for short) on-chip. The PosMap size is proportional to the number of data blocks (cachelines) in the memory and can be hundreds of megabytes. The solution to the above problem is recursive ORAM, proposed by Shi et al. [33]. The idea is to store the large data structures (PosMap in the Path ORAM case) in *additional* ORAMs to reduce the on-chip storage requirement. The cost of recursive ORAM is longer latency: now we have to access all the ORAMs in the recursion every time. Even after architectural optimizations [31], a recursive Path ORAM with reasonable configurations has to transferred over 400× data between the client and ORAM, compared with normal DRAM.

In this work, we propose two optimizations to recursive Path ORAM. First, we observe that there is locality in Path ORAM PosMap accesses. This means we can cache part of the PosMap in PosMap Lookaside Buffer (PLB) to reduce PosMap accesseses. Second, we propose a technique to compress Path ORAM PosMap using pseudorandom functions (PRF). However, utilizing PosMap locality or pseudorandomness turns out to be tricky: naïvely applying the ideas either leaks information on access patterns or hardly brings performance gain.

To address the problem, we propose *unified ORAM*. Unified ORAM can take advantage of PosMap locality and pseudorandomness in a provably secure fashion. Table 1 summarizes our improvements over recursive Path ORAM. Unified Path ORAM with PLB eliminates most of the PosMap accesses, reducing data movement from external memory by 45% compared with recursive Path ORAM, though the asymptotic performance stays the same. Pseudorandom PosMap leads to an asymptotic improvement over recursive Path ORAM,

Table 1: Asymptotic and empirical overhead of unified Path ORAM and recursive Path ORAM, in relation to normal memory. N is the total number data blocks. B_d is data block size. Parameter setting for the empirical results are given in Section 6.1.

ORAM scheme	Asymptotic	Empirical
Recursive [38]	$O\left(\log N + \frac{\log^3 N}{B_d}\right)$	460×
Unified + PLB	$O\left(\log N + \frac{\log^3 N}{B_d}\right)$	254×
Unified + PLB + PRF	$O\left(\log N + \frac{\log^3 N}{B_d \log\log N}\right)$	232×

and also slightly improves empirical performance. Combining the two ideas, unified ORAM outperforms recursive ORAM in both asymptotic and empirical performance. Our experiments show that using unified ORAM in a secure processor improves benchmark performance by about 60% on average.

The rest of the paper is organized as follows: Section 2 covers background on ORAM and (recursive) Path ORAM. Section 3 introduces the position map lookaside buffer, an important building block for our unified ORAM. Section 4 presents the architecture of the unified ORAM in detail. Section 5 introduces compressed PosMap using pseudorandom functions, and show it improves performance both asymptotically and empirically. We evaluate unified ORAM in Section 6, and discuss extensions in Section 7. Section 8 describes related work and Section 9 concludes.

2. BACKGROUND

In a system using ORAM, the programs send requests to an *ORAM controller*, and the ORAM controller interacts with the external untrusted storage (e.g., DRAM). Let A be the sequence of memory requests made by programs. Let R represent the observable interaction between the ORAM controller and the external storage to serve the memory request sequence A. ORAM requires that computationally bounded adversaries cannot learn anything about A from R. In other words, given any two program address sequences A_1 and A_2, the resulting interaction R_1 and R_2 should be indistinguishable (if we make them to have the same length).

In this paper, we focus on Path ORAM, but our idea works for several other ORAMs as well (see Section 8).

2.1 Basic Path ORAM

Path ORAM [38] is currently one of the most efficient ORAM schemes, and more importantly is practical for hardware implementation [25] due to its simplicity. Immediately after its first proposal in 2012 [35], it gained popularity in secure processors [5, 44, 25, 24].

In Path ORAM, the untrusted external storage is logically structured as a binary tree, as shown in Figure 1(a). We refer to it as **Path ORAM tree** or ORAM tree for short. The root of the ORAM tree is referred to as level 0, and the leafs as level L (we say such a tree has depth L). Each node in the tree is called a *bucket* and can hold up to Z data blocks (cachelines). Buckets that have less than Z data blocks are filled with *dummy blocks*. A dummy block is conceptually an empty slot that can be taken up by a real data block at any time. All the blocks in the tree including the dummy blocks are encrypted with probabilistic encryption (e.g., AES counter mode [22]), so any two blocks (dummy or real) are indistuiguishable after encryption. Each leaf node has a unique leaf label r. We also refer to the path from the root to leaf r as path r. The ORAM tree is not trusted and can be read by an adversary at any time.

The **Path ORAM controller** has a *position map*, a *stash* and associated control logic. The position map (PosMap for short) is a lookup table that associates each data block with a random leaf in the ORAM tree. The stash is a memory that stores up to a small number of data blocks that temporarily cannot be put back into the ORAM tree. We will assume the stash can store up to 200 blocks throughout the paper.

Path ORAM invariant and operation. At any time, each data block in Path ORAM is mapped to a random leaf via PosMap. Path ORAM maintains the following invariant: If a block is mapped to leaf r, then it must be either in some bucket on path r or in the stash.

The steps to access a block with program address a are as follows:

1. Look up PosMap with a, yielding the corresponding leaf label r.
2. Read all the buckets along path r into the ORAM controller. Decrypt all the blocks and add all real blocks to the stash (dummy blocks are discarded).
3. Due to the Path ORAM invariant, block a must be in the stash at this point. Return block a (to the last-level cache).
4. Assign a new random leaf r' to block a (update PosMap).
5. Evict and encrypt as many blocks as possible from the stash to path r while keeping the invariant. Fill any remaining space on the path with encrypted dummy blocks.

Path ORAM security. The path read and write operation (Step 2 and 5) are done in a data-independent way (e.g., always from the root to the leaf); and due to probabilistic encryption, all the ciphertexts along the path change. Therefore, the leaf label r is the only information revealed to an observer on an ORAM access. Step 4 is the key to Path ORAM's security, where a block is remapped to a new random leaf whenever it is accessed. This guarantees that PosMap always contains fresh random leaf labels for all the blocks, and thus a random path r, retrieved from PosMap, is read and written on every access regardless of the actual program address sequence.

2.2 Recursive Path ORAM

Recursive ORAM was first proposed by Shi et al. [33] to reduce client-side storage and was used to build Path ORAM in a secure processor setting [5, 31]. In our setting, the client is the secure processor and client-side storage must fit on-chip. The idea of recursive ORAM is that we can put the large client-side storage (PosMap for Path ORAM) into a *second* ORAM.

We refer to the original data Path ORAM as the *data ORAM* denoted as ORam_0, and the second Path ORAM as a *PosMap* ORAM denoted as ORam_1. Suppose each block in ORam_1 contains χ leaf labels for blocks in ORam_0. Then, for a block with address a_0 in ORam_0, its leaf label is in block $a_1 = a_0/\chi$ of ORam_1[1]. Accessing data block a_0 now

[1]All divisions in this paper are program disivion, a/χ meaning $\lfloor a/\chi \rfloor$. We omit the flooring for simplicity.

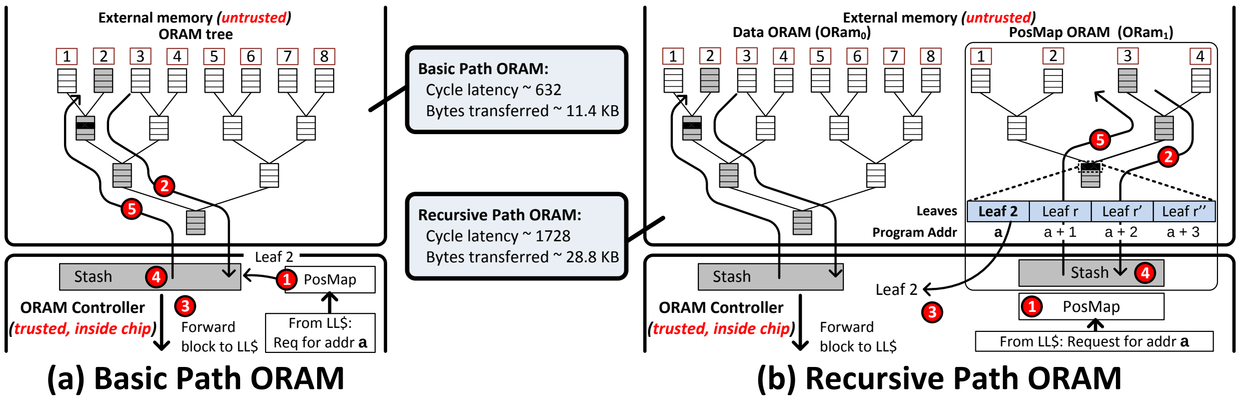


Figure 1: Basic and Recursive Path ORAM constructions. The addresses a, $a+1$, etc are tags of cachelines. In (b), a block in PosMap ORAM stores leaf labels for several cachelines. LL$ stands for Last-Level-Cache. The latency and the amount of data transferred are representative of the baseline recursive Path ORAM (see Section 2.3 and 6.1).

involves two steps. First, the ORAM controller accesses PosMap ORAM to get block $a_1 = a_0/\chi$, from which the ORAM controller retrieves the leaf label of a_0 and replaces it with a new random leaf label. Then it loads block a_0 from data ORAM. The steps in Figure 1(b) to access each ORAM align with Steps 1-5 from the basic Path ORAM case in Section 2.1.

Of course, the new on-chip PosMap might still be too large. In that case, additional PosMap ORAMs (ORam_2, ORam_3, $\cdots$, ORam_{H-1}) may be added to further shrink the on-chip PosMap. The PosMap blocks a_i ($i > 0$) are analogous to page tables in conventional virtual memory systems, where the leaf labels are pointers to the next-level page tables or the pages. Both page tables and PosMap ORAMs have to be queried on each main memory access; a recursive ORAM access is similar to a full page table walk.

2.3 (Recursive) Path ORAM Overhead

On each Path ORAM access, the ORAM controller has to read and write an entire path. We define $\mathsf{PathLen}$ to be the amount of data on a path. In an ORAM tree with depth L, each path contains $L+1$ buckets. Each bucket contains Z blocks and and some metadata for probabilistic encryption, e.g., the counter in AES counter mode [22]. Each block has B bits, and is stored alongside with its U_1-bit program address and U_2-bit leaf label. So the amount of data on a path (in bit) is

$$\mathsf{PathLen} = (L+1)[Z(B+U_1+U_2) + \mathsf{CounterSize}] \qquad (1)$$

The total amount of data movement in recursive ORAM is the sum of $\mathsf{PathLen}$ for all the ORAMs.

The $\mathsf{PathLen}$ metric directly relates to Path ORAM latency and energy overhead. Path ORAM latency and energy overhead are almost linear in $\mathsf{PathLen}$. The dominant part of Path ORAM latency is the time it takes to transfer $\mathsf{PathLen}$ Bytes of data between the external ORAM tree and the ORAM controller. Path ORAM energy mainly consists of three parts: (1) to access the external DRAM that stores the ORAM tree, (2) to transfer the data to and from ORAM controller and (3) to decrypt/encrypt the data. All three parts are linear in the amount of data movement.

In this paper, we care about both theoretical (asymptotic) and empirical (experimental) performance.

2.3.1 *Asymptotic performance*

Let N be the number of data blocks in ORAM, we need $U_1 = U_2 = \log N$. Now it makes sense to use counters also of $O(\log N)$ bits not to further increase complexity [2]. It is also reasonable to assume data block size B_d is at least $\Omega(\log N)$ in practice. Then Path ORAM's asymptotic bandwidth overhead is then given by $2 \cdot \mathsf{PathLen}/B_d = O(\log N)$.

For recursive ORAM, suppose each PosMap block contains χ leaf labels. Then PosMap block size is $B_p = \chi \log N$ After $H = O(\log N/\log \chi)$ levels of recursion, client-side PosMap size is reduced to $O(\log N)$. A good strategy, given in a more recent version of Path ORAM [39], is to make χ a constant (≥ 2). The resulting bandwidth overhead is then

$$\begin{aligned}\frac{2}{B_d}\Sigma_{h=0}^{H-1}\mathsf{PathLen_h} &= \frac{2}{B_d}O\left(B_d \log N + H \cdot B_p \cdot \log N\right)\\ &= O\left(\log N + \frac{\log^3 N}{B_d}\right) \qquad (2)\end{aligned}$$

2.3.2 *Empirical performance*

In the analysis of empirical performance throughout the paper, including Section 6, we assume $U_1 = U_2 = 32$ bits and $\mathsf{CounterSize} = 8$ bits. We assume a baseline recursive Path ORAM with 4GB capacity, data block size $B_d = 512$ bits, PosMap block size $B_d = 256$ (the best PosMap block size found by [31]) and $H = 5$. These parameters gives $\Sigma_{h=0}^{H-1}\mathsf{PathLen_h} = 14.4$ KBytes. This means 14.4 KBytes of data has to be read *and* written on each ORAM access, in contrast to a single access to a 64-Byte cacheline in normal memory. This is over $460\times$ data movement.

In fact, not very intuitively, more than half of the overhead comes from PosMap ORAMs. The $\mathsf{PathLen}$ for data ORAM is only 5.7 KBytes. The remaining 8.7 KBytes come from PosMap ORAMs. So all the PosMap ORAMs combined account for about 60% of the total amount of data movement, and accordingly about 60% of the latency and energy overhead. This paper addresses this issue and significantly reduces the overhead of PosMap ORAMs accesses.

[2]We remark that such counters will overflow in polynomial time. If that happens, all blocks are copied into a new Path ORAM and the algorithm resumes. This process takes $O(\log N)$ Path ORAM operations, a constant factor if amortized. This was implicitly assumed in previous work [38].

3. POSMAP LOOKASIDE BUFFER (PLB)

In this section, we introduce Position map Lookaside Buffer (PLB), which is an important building block of our unified ORAM.

When we introduced the recursive Path ORAM in Section 2.2, we intentionally compared it to virtual memory and position map to page tables. Conventional virtual memory systems have Translation Lookaside Buffers (TLB) to cache page tables. Similarly, PLBs aim at reducing the number of accesses to PosMap ORAMs by caching them on-chip. However, naïvely adding PLBs in recursive Path ORAM either breaks ORAM security or hardly gets any performance gain.

3.1 PLB Motivation

The key idea of PLB is that position map accesses have locality. A block in PosMap ORAM contains a set of leaf labels for *consecutive blocks* (see Figure 1(b)) in the next ORAM. Most programs have locality and tend to access neighboring cachelines most of the time, resulting in accessing the same PosMap block multiple times. Based on this insight, we propose using PLBs to cache PosMap ORAMs on-chip, to reduce the number of PosMap ORAM accesses.

As in Section 2.2, we refer to the data ORAM as ORam_0, the first PosMap ORAM as ORam_1, the second PosMap ORAM ORam_2 etc. On an ORAM access to address a_0, recursive Path ORAM has to access all the H ORAMs in the recursion in decreasing order (ORam_{H-1} first). For each PosMap ORAM ORam_i, we can add a PLB, referred to as PLB_i. Ideally, if block $a_i = a_0/\chi^i$ is already in PLB_i, the ORAM controller directly starts from accessing ORam_{i-1}, skipping ORam_i *and all the smaller PosMap ORAMs.* Unfortunately, this leaks information about the access pattern.

3.2 PLB (In)security

PLB hit/miss correlates directly to a program's access pattern—whether or not the access pattern has good locality. Consider the following simple example. Access pattern (1) scans each cacheline with a unit stride (i.e., it accesses cacheline 0, 1, 2, ...); Access pattern (2) uses a large stride such as 100 (i.e., it accesses cacheline 0, 100, 200, ...). Access pattern (1) will hit in PLB_1 most of the time and only accesses data ORAM. But access pattern (2) constantly misses in PLB_1 and needs to access ORam_1 everytime. Since ORam_0 and ORam_1 are stored at different physical locations in the external memory, an observer can clearly tell the two access patterns apart.

A simple fix to the above problem is to apply a public access rate each PosMap ORAM. Unfortunately, it is usually difficult to set the rates right, since the 'correct' rates depend on many factors like program locality, program input size, PLB capacity, PosMap ORAM block size and so on. Setting rates wrong may often negate all the benefits from PLB.

4. UNIFIED ORAM

We propose unified ORAM to fill the PLB security hole and extract substantial gain out of PLBs. Our key idea to fix the PLB insecurity is that, if we can make the accesses to any ORAM in the recursion indistinguishable, then we will not reveal PLB hit/miss results. We describe how unified ORAM works in Section 4.1, and then discuss how it makes PLB secure in Section 4.2.

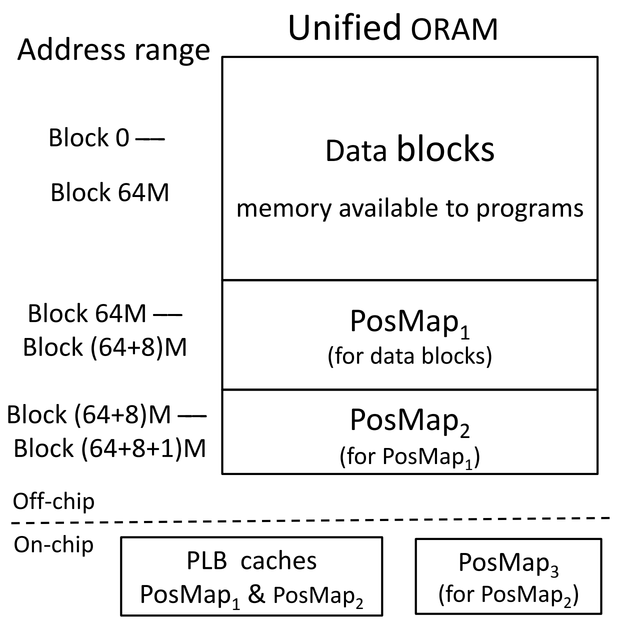


Figure 2: Unified ORAM address space. Assuming there are 64M data blocks, and each PosMap block contains $\chi = 8$ leaf labels for the next ORAM.

4.1 Address Mapping and Operations

In a unified ORAM, we still need hierarchical PosMap, denoted by $\{\mathsf{PosMap}_1, \mathsf{PosMap}_2, \cdots, \mathsf{PosMap}_{H-1}\}$. PosMap_1 is the PosMap for the data blocks (cachelines). PosMap_{h+1} is the PosMap for PosMap_h ($h \geq 1$), and will be smaller than PosMap_h by a factor of χ. There is only one Path ORAM tree, which contains all the data blocks as well as all the PosMap blocks (in $\{\mathsf{PosMap}_1, \mathsf{PosMap}_2, \cdots, \mathsf{PosMap}_{H-1}\}$). Data blocks and PosMap blocks must have the same size.

Different blocks occupy different logical address spaces in the unified ORAM, illustrated in Figure 2. Suppose there are N data blocks. They will occupy address space $[0, N)$. This is the memory space seen by the programs. Addresses beyond N are reserved for PosMaps and are not accessible to the programs. PosMap_1 occupies address $[N, N + N/\chi)$, PosMap_2 occupies address $[N + N/\chi, N + N/\chi + N/\chi^2)$, and so on (we assume N is an integer multiple of χ^H). The smallest PosMap (PosMap_2 in Figure 2) is stored on-chip.

For a data block with address a_0 ($a_0 < N$), its first-level PosMap block is the (a_0/χ)-th block in PosMap_1, which has address $a_1 = N + a_0/\chi$; its second-level PosMap block is the (a_0/χ^2)-th block in PosMap_2, which has address $a_2 = N + N/\chi + a_0/\chi^2$, and so on. Here is a simpler way to calculate the location of the leaf label for a block. For a block with address a_i [3], its leaf label is either in on-chip PosMap, or in block $a_{i+1} = N + a_i/\chi$ (which belongs to PosMap_{i+1}). It is easy to check the equivalence of the two approaches. Since $a_i = (\Sigma_{j=0}^{i} N/\chi^j) + a_0/\chi^i$,

$$\begin{aligned} a_{i+1} &= (\Sigma_{j=0}^{i+1} N/\chi^j) + a_0/\chi^{i+1} \\ &= N + [(\Sigma_{j=0}^{i} N/\chi^j) + a_0/\chi^i]/\chi \\ &= N + a_i/\chi \end{aligned} \qquad (3)$$

Note that all division includes flooring.

[3]it is a data block if $i = 0$, and belongs to PosMap_i otherwise

Below are the steps to access a data block with address a_0 ($a_0 < N$) in unified ORAM:

1. **(PLB lookup)** For $i = 1 : H-1$:

   (a) Look up the PLB for block $a_i = N + a_{i-1}/\chi$ (due to Equation 3).

   (b) If hit, let $h = i-1$ and go to Step 2; else, continue.

   If none of a_i hits, set $h = H-1$ and go to Step 2.

2. **(PosMap block accesses)** For $i = h : 1$ (i decreasing):

   Access the unified ORAM for block a_i and put it into the PLB. If this evicts another PosMap block from the PLB, add that block to the stash.

3. **(Data block access)** Access the unified ORAM for block a_0 and return it to the last-level cache.

In Step 2 and 3, the leaf label for block a_h is originally in PLB, or in the on-chip PosMap if $h = H-1$. The leaf labels for the other blocks a_i ($0 \le i < h$) are obtained from the unified ORAM tree, and blocks a_j ($1 \le j < h$) are brought into the PLB.

We will have only a single *unified* PLB, which can cache blocks from any PosMap and does not distinguish them in LRU replacement policy. We choose this design mainly for its simplicity. In addition, we believe this design is more robust to locality at different scale. For example, suppose accesses to PosMap_1 do not have good locality, and the PLB rarely hits on PosMap_1 blocks; then the LRU policy will tend to evict these PosMap_1 blocks, and effectively devote the space to caching other PosMap blocks. One can consider using more complicated PLBs, for example, having separate PLBs for different PosMaps, or favoring blocks from certain PosMaps based on their hit rate in the past. We leave these schemes to future work.

4.2 Security of Unified ORAM

Our security unified ORAM follows the security of Path ORAM. The basic operation of the unified ORAM is to load a block from the unified ORAM tree, which is exactly a Path ORAM operation—reading and writing a random path. Which block is accessed or whether it is a data block or a PosMap block is protected by Path ORAM. The security proof for Path ORAM [38] holds for unified ORAM.

We note that the original ORAM threat model assumes adversaries do not modify ORAM, and cannot monitor the ORAM timing channel. If we want to extend the threat model to defend against these two types of adversaries, we need additional techniques. We will discuss these techniques in Section 7.1, and show that unified ORAM outperforms recursive ORAM under all the threat models.

4.3 Hardware Simplifications

Besides improving ORAM performance and energy, another advantage of unified Path ORAM is that it reduces the hardware complexity of the ORAM controller. We compare the storage requirement of each major module for recursive ORAM and unified ORAM in Table 2.

The first simplication to hardware directly follows the fact that we now have only one ORAM tree. Thus we only need one stash, as opposed to one stash per ORAM in recursive Path ORAM. In the path write-back operation, blocks from any ORAM are treated the same: they are all ordinary blocks in the unified ORAM tree.

Table 2: The sizes of the modules inside the ORAM controller (in KBytes) for recursive ORAM and unified ORAM.

	Stash	PosMap	PLB	Total
Recursive ORAM $H = 4$	45	102	0	147
Recursive ORAM $H = 5$	45	5	0	50
Unified ORAM $H = 4$	14	32	32	78
Unified ORAM $H = 4$ with compressed PosMap	14	4	32	50

The on-chip PosMap of our unified ORAM (without compressed PosMap) is smaller than that of recursive ORAM with $H = 4$, but larger than that of recursive ORAM with $H = 5$. We will present in Section 5 compressed PosMap, which makes our on-chip PosMap as small as that of recursive ORAM with $H = 5$. We remark that for both recursive ORAM and unified ORAM, we can always add more levels of PosMaps in the recursion to further reduce the on-chip PosMap, at the cost of more data movement and longer ORAM access latency. So there is no good way to compare the on-chip PosMap size. But we also point out that unified ORAM makes it cheaper to add more PosMaps, because these extra PosMaps will usually be accessed less than 1% of time thanks to the PLB. As a comparison, in recursive ORAM, a sixth ORAM (though shorter in depth) still increases the latency by about 10%.

Another potential advantage of unified ORAM is to save the on-chip storage for leaf labels per cacheline. It is less relevant to our main results and requires some explanation, so we postpone its discussion to Section 7.2.

5. COMPRESSED POSMAP

In this section, we propose a technique to compress the position map of Path ORAM using Pseudo Random Function (PRF).

A PRF family $y = PRF_K(x)$ is a collection of efficiently-computable functions, where K is a random secret key. It guarantees that, anyone who does not know K (even given x) cannot distinguish y from a truly random bit-string in polynomial time with non-negligible probability [9].

5.1 Construction

Following our notation and analysis in Section 4.1, suppose each PosMap block contains k leaf labels for the next ORAM. Then the leaf labels for block $\{a,\ a+1,\ \cdots,\ a+\chi-1\}$[4] will be retrieved from block $a/\chi + N$ (due to Equation 3). Block $a/\chi + N$ contains contains a α-bit group counter (GC) and χ β-bit individual counters (IC):

$$GC \mid\mid IC_0 \mid\mid IC_1 \mid\mid IC_2 \mid\mid \cdots \mid\mid IC_{\chi-1},$$

where $||$ is concatenation. The leaf label of block $a+j$ is defined to be

$$PRF_K(GC \mid\mid IC_j \mid\mid a+j) \mod 2^L.$$

The output (we will call it uncompressed format) is computationally indistinguishable from a truly random number in

[4] a is a multiple of χ here so that these χ blocks share the same PosMap block. The addresses $a, a+1, \cdots$ can be either data blocks or PosMap blocks, so we omit the subscript i.

$[0to2^L)$ as long as we never feed the same input to the PRF. Block identifier $a+j$ is included in the input, so different blocks will always use different inputs to the PRF. If we further ensure that the concatenation of the group counter and the individual counter strictly increases, then the PRF always receives new input. This is achieved by the following modified remapping operation.

When remapping block $a+j$, the ORAM controller first increments its individual counter IC_j. If the individual counter overflows (becomes zero again), the ORAM controller will increment the group counter GC. This will change the leaf label for all the blocks in the group, so we have to load all of them from their current paths, reset all the individual counters and remap them to their new paths given by $PRF_K(GC+1 \mid\mid 0 \mid\mid a+j) \mod 2^L$. In the worst case where the program always requests the same block in a group, we need to reset all the χ individual counters in the group every 2^β accesses.

This would be very expensive in recursive ORAM In that case, ORAM controller has to make χ full recursive ORAM accesses to reset individual counters in *a certain* ORAM. Otherwise, it reveals that individual counters have overflown in that certain ORAM, which is related to the access pattern. As in the case of PLB, unified ORAM helps by making an access for resetting an individual counter indistinguishable from an normal unified ORAM access. So the worse case amortized cost to reset individual counters is $\chi/2^\beta$ for unified ORAM.

We note that each block in the ORAM tree or in the stash also has its own leaf label stored beside itself. These leaf labels will still be in the uncompressed format, because they are used in path write-back, where the ORAM controller does not have a chance to reference the PosMap.

5.2 Asymptotic Improvement

Compressed PosMap can lead to an asymptotic improvement by reducing the levels of recursion needed. We choose group counter width to be $\alpha = \Theta(\log N)$, the same length as the counters for probabilistic encryption, and group counter overflow can be handled in the same way. Let individual counter width to be $\beta = \log\log N$, and $\chi = \frac{\log N}{\log\log N}$. Then PosMap block size is $B_p = \alpha + \chi \cdot \beta = \Theta(\log N)$. Note that such a compressed PosMap block contains $\chi = O\left(\frac{\log N}{\log\log N}\right)$ leaf labels. Without compression, a PosMap block of size $O(\log N)$ can only contain a constant number of leaf labels. Such a parameter setting reduces the levels of recursion $H = O\left(\frac{\log N}{\log \chi}\right)$ by a factor of $\log \chi = O(\log\log N - \log\log\log N) = O(\log\log N)$, and only introduces a constant amortized overhead of resetting individual counters as $\chi/2^\beta = O(1)$.

Now we study the asymptotic performance of unified ORAM with compressed PosMap[5]. We start by examining a simple case where data block size $B_d = B_p = \Theta(\log N) = B$. Then we can have a unified Path ORAM with block size B. For each ORAM access, we need to access the unified ORAM H times (walk the PosMap), and each time transfers $O(B \cdot \log N)$ bits. So the asymptotic overhead is

$$\frac{2}{B}(B \cdot \log N) \cdot H = O\left(\frac{\log^2 N}{\log\log N}\right).$$

[5]When analyzing asymptotic performance, we assume PLB never hits, since there is no good way to model locality.

Table 3: Asymptotic overhead of unified Path ORAM and previous ORAM schemes. N is the total number data blocks. B_d is data block size. $B_d = \Omega(\log N)$ is requried for Kushilevitz et al..

ORAM scheme	Asymptotic
Kushilevitz et al. [18]	$O\left(\frac{\log^2 N}{\log\log N}\right)$ $(B_d = \Omega(\log N))$
Recursive Path ORAM [38]	$O\left(\log N + \frac{\log^3 N}{B_d}\right)$
Unified Path ORAM	$O\left(\log N + \frac{\log^3 N}{B_d \log\log N}\right)$

This bound outperforms that of recursive Path ORAM by a factor of $O(\log\log N)$, and breaks even with Kushilevitz et al. [18] (currently the best ORAM scheme with small client storage under small block size) for $B_d = \Theta(\log N)$.

For the general case where $B_d \neq B_p$, we are faced with the problem of choosing unified ORAM block size. Unified ORAM should still use block size B_p, because a larger block size for PosMap is suboptimal [31, 39]. Then we need to break each data block into sub-blocks of size B_p and store them in unified ORAM as independent blocks. However, we let these sub-blocks share a single individual counter; the uncompressed leaf for each sub-block is retrieved by including sub-block index in the PRF input, $PRF_K(GC \mid\mid IC_j \mid\mid a+j \mid\mid k) \mod 2^L$. Now a full ORAM access involves H accesses to unified ORAM to load the above PosMap block, and another $\lceil B_d/B_p \rceil$ accesses to unified ORAM to load all the sub-blocks of the data block. The asymptotic performance is

$$\frac{2}{B_d}(B_p \cdot \log N) \cdot \left(\left\lceil \frac{B_d}{B_p} \right\rceil + H\right) = O\left(\log N + \frac{\log^3 N}{B_d \log\log N}\right).$$

This bound is also asymptotically better than that of recursive Path ORAM when $B_d = o(\log^2 N)$. When we have large data block size $B_d = \Omega(\log^2 N)$, recursive and unified ORAM both achieve $O(\log N)$ asymptotic performance—the ORAM lower bound [10]—and outperform Kushilevitz et al. [18].

The above results are summerized in Table 3. Unified Path ORAM has the best asymptotic overhead under any $\Omega(\log N)$ data block size among ORAMs with small client storage.

5.3 Practical Improvement

Currently, ORAMs in secure processors or other applications is usually not large enough to see the performance improvement of our theoretical contruction above. But compressed PosMap still has benefits in practice. First, it reduces the final on-chip PosMap size. Second, it allows us to cache more leaf labels in PLB, and to bring more leaf labels on every PosMap block access. The latter helps increase PLB hit rate and reduces data movement.

Concretely, we will choose $\alpha = 64$ (the same as CounterSize and never overflows), $\beta = 14, \chi = 32$ forming a 64-Byte block. With the above parameters, we are able to fit 32 leaf labels in each 64-Byte PosMap block. Effectively, each leaf label is now only 16-bit, and the overhead of resetting individual counters is at most $\chi/2^\beta = 0.2\%$. Originally, each leaf label in the uncompressed format must be at least $\log N$ bits (≈ 26).

The compressed PosMap does not lower the security level of Path ORAM or add additional hardware. Even without compressed PosMap, a PRF will be used to generate the fresh

Table 4: Processor Configuration

Core, cache and DRAM	
number of cores	4
core model	in order, single issue, 1GHZ
add/sub/mul/div	1/1/3/18 cycles
fadd/fsub/fmul/fdiv	3/3/5/6 cycles
L1 I/D cache	32 KB per core, 4-way, LRU
L1 data + tag access time	1 + 1 cycles
shared L2 Cache	512KB per core, 8-way, LRU
L2 data + tag access time	8 + 3 cycles
cacheline size	64 B
DRAM latency	50 cycles
DRAM bandwidth	20 GB/s
Path ORAM configuration	
data ORAM capacity	4 GB
data block size	64 B
blocks per bucket (Z)	3
ORAM latency model	See Section 6.1
on-chip storage	See Table 2
Recursive Path ORAM	
PosMap block size	32 B
number of ORAMs (H)	5 (4)
latency	1728 (1524) cycles
Unified Path ORAM	
PosMap block size	64 B
number of logical ORAMs (H)	4
latency (1 access)	632 cycles
PLB	32 KB, 4-way, LRU

Table 5: Input and Argument for SPLASH-2 Benchmarks

Bench	Input Set A	Input Set B
barnes	16K particles	256K particles
fmm	16K particles	256K particles
cholesky	tk29.O	tk29.O
fft	2^{20} complex numbers	2^{24} complex numbers
lu_c	1024 × 1024 matrices	2048 × 2048 matrices
lu_nc	768 × 768 matrices	1024 × 1024 matrices
ocean_*	258 × 258 grid	514 × 514 grid
radix	4M keys, radix = 4K	64M keys, radix = 4K
raytrace	Car	Balls4, -a2
volrend	head, Rotate Step = 8	head, Rotate Step = 50
water-ns	8^3 particles	15^3 particles
water-s	8^3 particles	32^3 particles

leaf labels. In practice, we will use AES as PRF. Compressed PosMap, however, does add an additional AES latency to the critical path. In the PosMap lookup step, the ORAM controller has to perform an AES operation (evaluate the PRF) to get the uncompressed random leaf. This overhead is small compared with the hundreds of cycles data latency.

6. EVALUATION

6.1 Methodology

We evaluate our unified ORAM using the Graphite simulator [27]. The processor configurations are listed in Table 4. The core and cache model remain the same in all experiments.

We use SPLASH-2 [34] and a representative subset of SPEC06 integer [16] benchmarks. For SPLASH-2 benchmarks, we have two sets of inputs and arguments listed in Table 5. For SPEC benchmarks, we use their reference inputs.

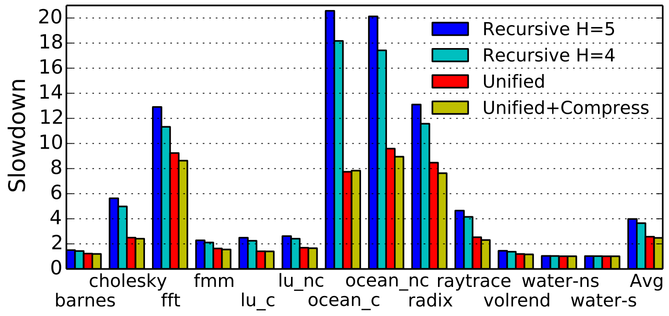


Figure 3: Completion time on SPLASH-2 benchmarks with input set A. Results are normalized to the insecure baseline.

All the SPLASH-2 benchmarks are run to completion, and all the SPEC benchmarks are run for 5 billion instructions. SPLASH-2 with input set B and 5 billion instructions for SPEC are the limit of our simulation capability. Thus we will only report results for them in Section 6.2 to show the benefit of unified ORAM generalizes to larger problem size and diverse benchmarks. For all the other studies, we use SPLASH-2 with input set A.

We evaluate the four ORAMs in Table 2, and report benchmark slowdown compared with a conventional system that uses DRAM. Recursive ORAMs use 32-Byte PosMap block size following [31], but we changed data block size from 128-Byte to 64-Byte to be more close to current processors. The one with $H = 5$ uses the same amount of on-chip storage as our unified ORAM with compressed PosMap, so we will take it as the baseline recursive Path ORAM.

We assume ORAM latency consists of data latency and a 50-cycle encryption latency following [31]. The memory layout [31] enables data to be transferred at over 93% of DRAM peak bandwidth between the ORAM controller and the ORAM tree. We assume an effective bandwidth $\mathsf{BW} = 20$ GB/s, which is roughly the bandwidth of two DDR3 memory channels.

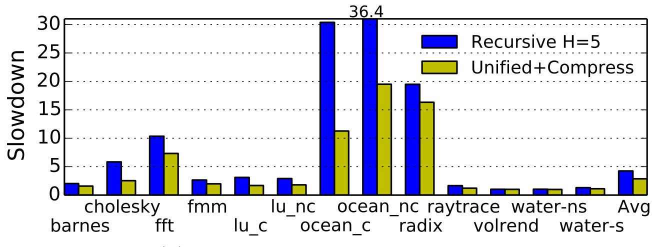
6.2 Performance Improvement

Figure 3 shows the performance improvement of unified Path ORAM over recursive Path ORAM on SPLASH-2 benchmarks with input set A. The performance in the figure is normalized to the insecure baseline, which uses DRAM. On average (geometric mean), unified Path ORAM without compressed PosMap improves performance by 53% compared with the baseline recursive Path ORAM with $H = 5$. Compressed PosMap leads to another 6% performance improvement, achieving 1.61× speedup over the baseline recursive Path ORAM. Compared with recursive ORAM with $H = 4$, which uses about 100 KB more on-chip storage, unified ORAM still has 42% performance gain.

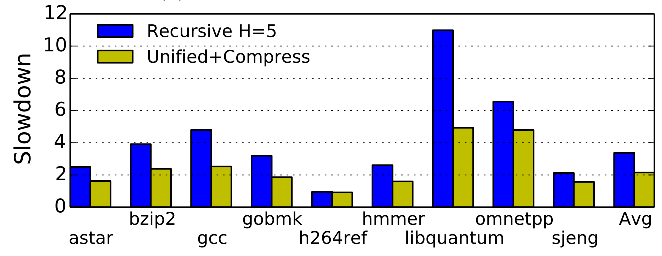
Figure 4 demonstrates that unified ORAM also has a significant improvement on SPLASH-2 with input set B and SPEC benchmarks. The speedup over recursive ORAM is 1.48× and 1.57×, respectively.

6.3 PLB Hit Rate and Energy Improvement

As mentioned in Section 2.3, Path ORAM latency and energy overhead is linear in the amount of data transferred between the ORAM controller and the ORAM tree. We now



(a) SPLASH-2 with input set B



(b) SPEC with reference inputs

Figure 4: Completion time on SPLASH-2 with input set B and SPEC with reference inputs. Results are normalized to the insecure baseline.

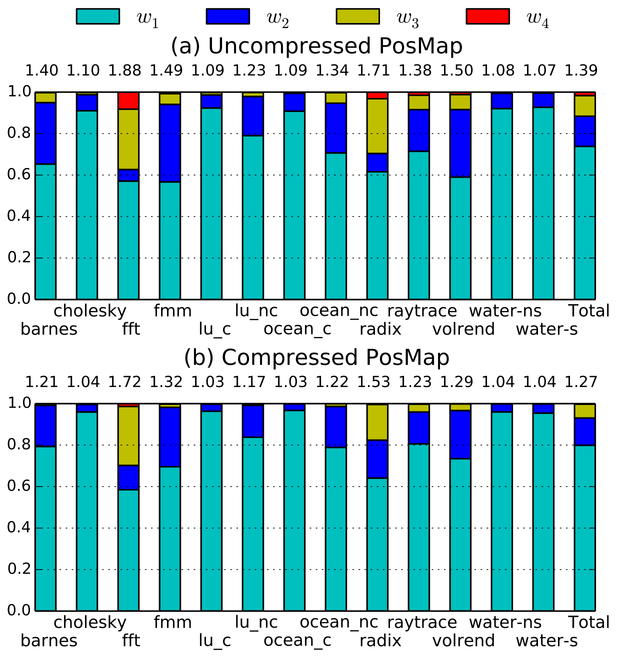


Figure 5: Distribution of the number of accesses in unified ORAM with and without compressed PosMap.

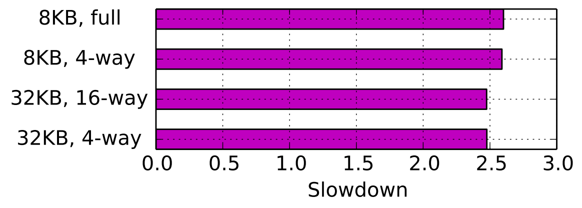


Figure 6: SPLASH-2 performance (geometric mean) on input set A with different PLB capacity and ways.

analyze the average data movement of unified ORAM and compare it with the recursive ORAM.

We make the unified ORAM tree have the same parameters as the data ORAM tree in recursive ORAM. So PathLen of the unified ORAM tree is equal to that of the data ORAM tree in recursive ORAM, both being 5.7 KBytes. Let w_i ($1 \le i \le H$) be the probability that we need to access unified ORAM i times on an access. The average data movement of the unified ORAM per access is $DM_U = 2 \times \mathsf{PathLen} \cdot \Sigma_{i=1}^{H} i \cdot w_i$. The coefficient 2 comes from the fact we have to read *and* write the path on each access. Figure 5 shows the dsitribution of w_i. The number on top of each bar is the weighted sum $\Sigma_{i=1}^{H} i \cdot w_i$.

As expected, the benchmarks with good locality (e.g., cholesky, ocean_contiguous and lu_contiguous) are the ones that benefit most from unified ORAM. For them, unified ORAM with PLB almost eliminates all the accesses to PosMap blocks (weighted sum close to 1). Most of the benchmarks already have very few accesses to PosMap blocks, which is why the improvement of compressed PosMap is small across all benchmarks. FFT and radix have relatively worse locality with $w_1 \approx 0.6$, and compressed PosMap improves their performance by over 10%. Though water_* also have good locality, they rarely access main memory, so they have almost the same performance with unified ORAM, recursive ORAM and even DRAM. No benchmark in our experiments has too poor locality that makes unified ORAM worse than recursive ORAM.

On average, without compressed PosMap, $DM_U = 2.78 \times$ PathLen; with compressed PosMap $DM_U = 2.54 \times$ PathLen. Compared with the baseline recursive Path ORAM where $DM_R = 2 \times 14.4$ KBytes $\approx 5 \times$ PathLen, unified ORAM with compressed PosMap reduces data movement by 49%. The almost 2× saving on data movement also means a nearly 2× reduction in ORAM energy comsumption.

6.4 PLB Capacity and Ways

Figure 6 gives the average performance of SPLASH-2 benchmark on input set A with different PLB capacity and set-associativity. A smaller PLB (8KB) hurts performance by 5% on average, but on some benchmarks the performance loss is over 10% (not shown in the figure). Increasing PLB set-associativity does not bring observable benefits.

7. EXTENSIONS

7.1 Extending the Threat Model

We now discuss the techniques to defend against timing attacks or active adversaries and show that unified ORAM outperforms recursive ORAM under different threat models.

Integrity requires that adversary cannot tamper with the data in ORAM. In other words, the program always gets from memory what it wrote to that location last time. We use the scheme in [30], which is a variant of a Merkle tree [26] that efficiently takes advantage of the Path ORAM tree structure. The overhead is that 2 hashes per bucket need to be stored in the ORAM tree and read/updated on each access. We assume a SHA-1 hash [4], which is 20 Bytes is added to each bucket in Path ORAM tree.

The timing of memory accesses can also leak the access pattern. For the most simple example, the better locality an access pattern has, the more it hits in on-chip cache, and

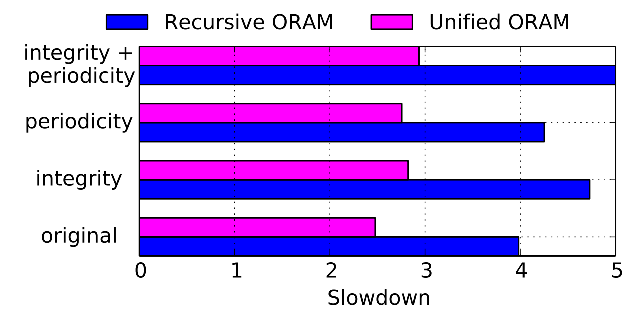


Figure 7: Normalized SPLASH-2 benchmark performance on input set A (geometric mean) under different threat models.

the less infrequently it accesses ORAM. Ascend proposed periodically accessing the main memory to protect the timing channel [5, 6]. In their proposal, the ORAM controller serves the next ORAM request $OInt$ cycles after the previous one finishes. If no real request comes within $OInt$ cycles after the previous one, the ORAM controller makes a dummy access (to a reserved block that contains no useful data). We use $OInt = 100$.[6]

Figure 7 compares the average slowdown on SPLASH-2 with input set A when using recursive and unified Path ORAM under different threat models. From left to right are the original ORAM threat model, with integrity verification alone, with periodicity alone and with both. Unified Path ORAM has significant performance gains over recursive Path ORAM in all the four cases, with the speedup being 61%, 68%, 54% and 71%, respectively.

7.2 Not Storing Leaf Labels with Cachelines

Both previous work [31] and this paper thus far assumed that when a block is evicted to the ORAM controller, it can be directly appended to the stash without accessing any path. A block in the stash must have its leaf label with it; otherwise, the ORAM controller does not know which path it should put this block to. So the above assumption requires adding a leaf label field to *every* block in PLB and on-chip cache. Assuming 64-Byte blocks and 4-Byte leaf labels, it introduces 1/16 storage overhead. While this may be fine for PLB (since PLB is small), it is a significant overhead to on-chip cache (128KB for a 2MB cache). In addition to the extra storage, it also requries modification to the current cache design, making it more difficult to integrate ORAM with processors.

If leaf labels are not stored in on-chip cache, then for every *data* block evicted from last-level cache, the ORAM controller has to access the PosMap ORAMs to retrieve its current leaf label. In recursive ORAM, data ORAM must be accessed as well for security, resulting in up to 2× data movement. Again, PLB and unified ORAM come to help in this case. Figure 8 shows that the performance penalty of not storing each cacheline's leaf label on-chip is 60% for recursive ORAM, and only 15% for unified ORAM.

7.3 Relation to Virtual Memory and TLB

Position map and PLBs in Path ORAM are similar to page

[6]The ORAM controller is idle for $OInt$ cycles after each access. Since a unified ORAM access takes less time than a recursive ORAM access, unified ORAM consumes less power than recursive ORAM under the same $OInt$.

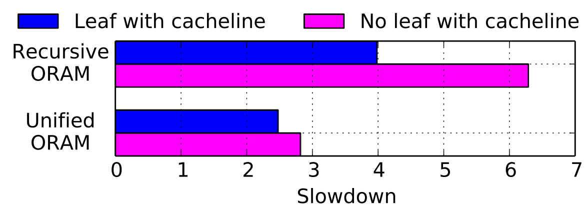


Figure 8: The performance penalty (geometric mean) on SPLASH-2 benchmarks with input set A of not storing cacheline's leaf label on-chip.

tables and TLBs in conventional virtual memory systems. A page table converts virtual addresses to physical addresses, while a PosMap associates program addresses with leaf labels, which represent a set of (randomized) physical addresses which contains the data of interest. A PLB caches PosMap, in the similar way a TLB caches page tables.

However, PosMap in unified or recursive ORAM is still very different from the virtual memory system. The primary motivation of virtual memory is to provide program memory separation and to allow programs to use more memory than what is physically available (through paging). Were it not for these two reasons—consider a machine that runs only one program at a time and has no disk—virtual memory has no reason to exist. Yet even in that case, PosMap is a necessary module of Path ORAM. In addition, virtual memory and page tables (and TLB in some systems) are managed by the operating system (OS). In contrast, a PLB is accessible only to the unified ORAM controller, and is transparent to any software (including the OS).

Virtual memory and page tables can be added on top of unified ORAM. In such a system, a virtual address is first translated into a real address via the page table; the real address is then translated into a set of randomized physical addresses (represented by a random leaf) via the PosMap. The entire virtual memory space including the page tables have to reside in data ORAM address space $[0, N)$. TLBs can be added to cache page tables (note again page tables are part of the data ORAM). ORAM address space beyond N is reserved for PosMap ORAMs and is cached in PLBs. If we have "oblivious disk", we can also support paging securely, but that is outside the scope of this paper.

8. RELATED WORK

8.1 ORAM Algorithms

Since ORAM's first proposal in 1987 [8], there has been significant follow-up work that has resulted in more efficient ORAM schemes [28, 10, 29, 13, 18, 43, 11, 12, 33, 37].

Though we focused on Path ORAM [38] in this paper, our ideas also apply to some other ORAM constructions [33, 37, 7] that have large client-side storage, proportional to the size of the memory. If these ORAMs are to be used in secure processors, they all need recursion to reduce client-side storage, and unified ORAM applies to them as well.

8.2 ORAM Optimization and Implementation

Ren et al. [31] explored the (recursive) Path ORAM design space and proposed several optimizations to (recursive) Path ORAM. We use their optimized recursive Path ORAM as the baseline in our work and significantly improve performance and energy consumption.

Lorch et al. [23] exploited the parallelism in ORAM operations and used multiple trusted coprocessors to speedup ORAM accesses. Our work is orthogonal to theirs. We try to reduce the total amount of work an ORAM has to do, while they parallelized the ORAM operations. Our unified ORAM also has substantial parallelism, and can adopt their techniques. For the same reason, we can also saturate much higher bandwidth than what we assumed in this work like previous works [23, 31, 25].

Phantom [25] is the first hardware implementation of ORAM. The authors implemented a single Path ORAM without recursion on an FPGA, partly due to the additional complexity of recursive Path ORAM. As a result, the design is not scalable, and it had to use a very large block size (4KB) to reduce the total number of blocks (and hence PosMap size). Our unified ORAM can easily support any realistic block size and memory capacity. Compared with recursive ORAM, unified ORAM is easier to implement, consumes less area and provides much better performance.

8.3 Virtual Memory and TLB

Virtual memory may have first been proposed by Fritz-Rudolf Güntsch in his PhD thesis, and was later used in Atlas computer [17]. Hatfield and Gerald [15] identified locality in page table accesses, which inspired the use of TLBs [32, 3]. We do get some inspiration from TLBs, and we use the name PLB in tribute to the early contributors of virtual memory and TLBs. In a sense, we borrow well-established architectural ideas and apply them to improve Path ORAM, a state-of-the-art cryptographic primitive. We hope that our work is viewed as an example of classical architectural ideas continuing to contribute in new frontiers.

9. CONCLUSION

This paper identifies locality in position map of Path ORAM, and proposes compressing the position map with pseudorandom functions. Both techniques require the use of unified ORAM, where all the logical ORAMs are stored in the same Path ORAM tree. Our optimizations achieve both asymptotic and empirical improvement. Experiments show that unified ORAM with compressed position map reduces ORAM data movement by half and improves SPLASH-2 and SPEC performance by 61% and 57%, respectively.

We are currently building a processor with unified Path ORAM and PLB. Compared with previous work, we expect our unified ORAM to have similar hardware complexity, but be able to support any reasonable block size and memory capacity, and at the same time significantly reduce ORAM latency and energy overhead.

10. REFERENCES

[1] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[2] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *Advances in Cryptology–EUROCRYPT 2013*, pages 279–295. Springer, 2013.

[3] D. W. Clark and J. S. Emer. Performance of the vax-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems (TOCS)*, 3(1):31–62, 1985.

[4] D. Eastlake and P. Jones. RFC 3174: US secure hashing algorithm 1 (SHA1), Sept. 2001.

[5] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing*, pages 3–8, Oct. 2012.

[6] C. W. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. 2014.

[7] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2013.

[8] O. Goldreich. Towards a theory of software protection and simulation on oblivious rams. In *STOC*, 1987.

[9] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[10] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *J. ACM*, 1996.

[11] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM.

[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 13–24, New York, NY, USA, 2012. ACM.

[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[14] D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing.* Intel Press, 2006.

[15] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[16] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[17] T. Kilburn, D. B. Edwards, M. Lanigan, and F. H. Sumner. One-level storage system. *Electronic Computers, IRE Transactions on*, (2):223–235, 1962.

[18] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[19] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[20] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 178–192, 2003.

[21] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[22] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, Maryland, USA, 2000.

[23] J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman. Toward practical private access to data centers via parallel oram. *IACR Cryptology ePrint Archive*, 2012:133, 2012. informal publication.

[24] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. A high-performance oblivious ram controller on the convey hc-2ex heterogeneous computing platform. 2013.

[25] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. ACM CCS, 2013.

[26] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[27] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, 2010.

[28] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[29] B. Pinkas and T. Reinman. Oblivious ram revisited. In *Advances in Cryptology–CRYPTO 2010*, pages 502–519. Springer, 2010.

[30] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *Proceedings of the 17th IEEE High Performance Extreme Computing Conference*, September 2013.

[31] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the Int'l Symposium on Computer Architecture*, June 2013. Available at Cryptology ePrint Archive, Report 2012/76.

[32] M. Satyanarayanan and D. Bhandarkar. Design trade-offs in vax-11 translation buffer organization. *Computer*, 1981.

[33] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, pages 197–214, 2011.

[34] J. P. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, 1992.

[35] E. Stefanov and E. Shi. Path O-RAM: An Extremely Simple Oblivious RAM Protocol. Cornell University Library, arXiv:1202.5150v1, 2012. arxiv.org/abs/1202.5150.

[36] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[37] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[38] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the ACM Computer and Communication Security Conference*, 2013.

[39] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. ACM CCS, 2013. Available at Cryptology ePrint Archive, Report 2013/280.

[40] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *ICS (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[41] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32^{nd} *ISCA'05*, New-York, June 2005. ACM.

[42] Trusted Computing Group. TCG Specification Architecture Overview Revision 1.2. http://www.trustedcomputinggroup.com/home, 2004.

[43] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 293–304, New York, NY, USA, 2012. ACM.

[44] X. Yu, C. W. Fletcher, L. Ren, M. van Dijk, and S. Devadas. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*, 2013.

[45] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th ASPLOS*, 2004.